M16C/62

Using the M16C/62 UART for SPI[1] Communication

1.0 Abstract

This article presents an example that uses a UART on an M30624 MCU as an SPI-compatible master device. SPI (Serial Peripheral Interface) is a synchronous communications format used by many peripheral devices. Although multiple slave devices may be connected to a single SPI master, for clarity, the circuit in this example has only one slave device.

As in any application, please ensure that you have an adequate understanding of the external device before connecting it to the M16C/62's I/O pins.

2.0 Introduction

The Renesas M30624 has three independent UARTs, each of which provides full duplex, serial communication with external devices, including other MCUs. "USART" would be more appropriate because communication may be synchronous or asynchronous. Each mode supports several data formats. Format settings in asynchronous mode include 7, 8, or 9 bits per character; even, odd, or no parity; and 1 or 2 stop bits. In synchronous mode, transfers are always 8 bits per character with selectable msb- or lsb-first and clock polarity settings. A UART in synchronous mode may serve as the master SPI device in a single-master configuration, depending on the requirements of the slave device(s).

More information about synchronous mode operation of the UART may be found in section 2.4, "Clock-Synchronous Serial I/O," of revision C.4 of "M16C/62 Group: User's Manual.

3.0 Contents

The M16C/62 UART was designed to be flexible and easy to use. The SPI protocol is a superset of its synchronous mode features. Thus, there are some limitations on its compatibility.

3.1 SPI Formats

A basic SPI interconnection consists of four wires:

- SDI, serial data input
- SDO, serial data output
- SCK, serial clock
- SS´, slave select (active low)

[1] SPI is a trademark of Motorola, Inc.

The directions of SDI and SDO are relative to the master device. The master generates SCK and a separate SS´ to each slave. SS´ is an active-low enable. Because SCK may idle high or low and the transmitters may be clocked by its rising or falling edges, the four possibilities for the data-to-clock relationship are shown in Figure 1.

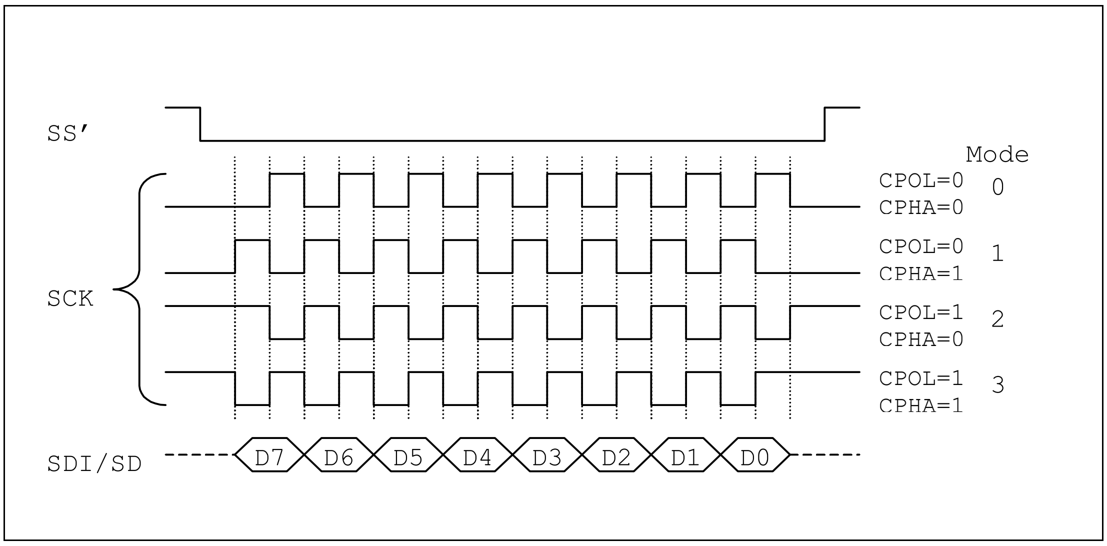

Figure 1 SPI Clocking Formats (msb-first data shown)

The data may be transferred lsb-first, instead of msb-first. CPOL and CPHA refer to settings for the clock polarity and clock phase, respectively, as described in the SPI protocol. Some manufacturers of peripheral ICs use a shorthand "(CPOL,CPHA)" notation, as in "(0,0)", while others refer to supporting a particular "SPI Mode," as in "SPI Mode 0."

3.2 An M16C/62 MCU as an SPI Master

The basic SPI interconnection described above is implemented as follows:

- RXDi serves directly as SDI.
- TXDi serves directly as SDO.
- SCLKi serves directly as SCK.
- An output port serves as SS´.

where i = 0, 1, or 2, depending on the UART selected. The example circuit in this article uses UART2 for communication and P73 as SS´. P73 must be toggled by software for proper operation.

Bit 7 of UiC0, UFORM, selects lsb-first (0) or msb-first (1) transfers. The polarity of SCLKi is selectable via bit 6 of UiC0, CKPOL; note, however, the sense of CKPOL is inverted from that of the protocol's CPOL. The clock phase cannot be altered and is always 1. Thus, CKPOL=0 selects (1,1) or SPI Mode 3, while CKPOL=1 selects (0,1) or SPI Mode 1.

3.3 25C160 EEPROM

The 25C160 is a 2Kx8 EEPROM with an SPI interface. Several manufacturers offer this EEPROM, including Atmel (AT25160 (-2.7)), Fairchild (NM25C160), Microchip (25AA160/25C160/25LC160), ST Microelectronics (M95160 (-W, -S)), and Xicor (X25160). A 25C160 by Microchip was used in developing the example in this article.

Communication is half-duplex only, msb-first data, and SPI Mode 0 or 3. The chip contains a small state machine and a status/control register. There are six instructions for controlling the state machine: set or reset the write enable latch in the status register; read or write the status register; and read from or write to the memory array. Reads from the array may begin at any address and be of any number of consecutive bytes. Writes to the array may begin at any address but must be of 16 bytes or less, all of which must reside in the same memory "page" (that is, address bits 11-4 do not change).

Each operation begins with the MCU driving the CS´ pin low and then transmitting the one-byte instruction. The MCU transmits a second byte if writing to the status register or receives a byte if reading from it, or transmits two bytes of address if accessing the array. If writing to the array, the data are transmitted, or, if reading from the array, the desired number of bytes is received. Each operation ends with CS´ driven high. Following a write to the array or to the status register, the Write-In-Progress bit of the status register is 1 while the write operation is being performed.

More details about the device are available from the respective manufacturer.

3.4 Implementation

For this example, the interconnection of MCU and EEPROM is shown in Figure 2. Omitted are such details as power supplies, Xin crystal, reset generation, and so forth. The 2.4KΩ pull-up resistor is required since P70 is an open-drain driver; it would not be needed if UART0 or UART1 were used instead of UART2. No pull-up is required on P71 since it is an input in this circuit.

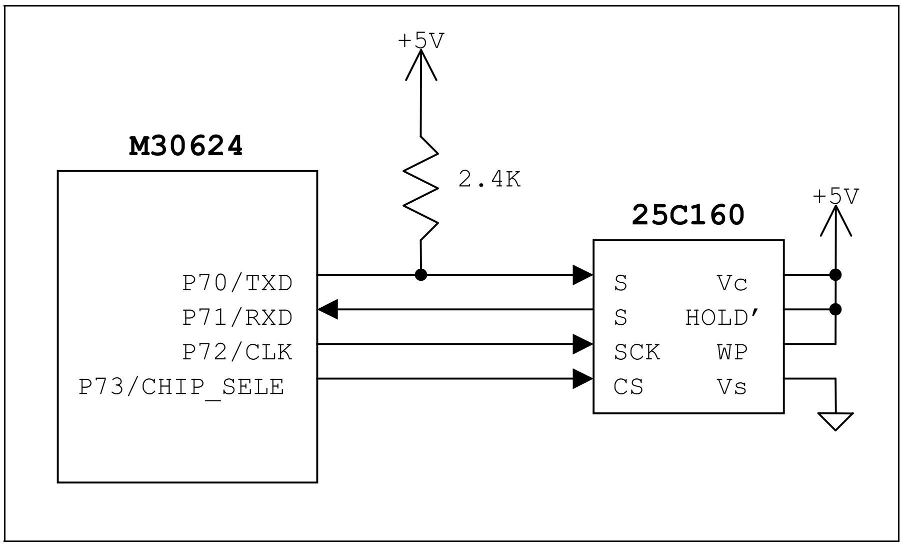

Figure 2 Circuit Fragment

Development of this example used an MSV1632 Starter Kit with an M30624 MCU operating at 5 volts and 16 MHz. The interaction between the program and the EEPROM is entirely request/response, so the UART is polled instead of interrupt- or DMA-driven. Although the program could be made slightly more efficient by reading the EEPROM's status byte at the end of each interaction instead of calling EE_read_status(), the small delays between routines allow the EEPROM's timing requirements to be met transparently.

Despite having SPI Mode 3 in common, the handshaking between the M16C/62 UART and the 25C160 EEPROM is not as seamless as its datasheet might imply: At the end of each interaction, the SCK input must be driven low when CS´ transitions high. When the UART is idle, CKPOL (bit 6 of register U2C0) gives direct control of the CLK2 output. As mentioned above, CKPOL selects compatibility between SPI Modes 1 and 3, so it must be returned to the proper setting after each direct-control use

Caution: When using CKPOL for direct control of the CLK pin, disable the UART's transmitter and receiver before altering CKPOL to avoid unpredictable behavior. Also, check that the direct control of CLK will not adversely affect the operation of devices connected to this pin.

When the UART is in synchronous mode, reception is simultaneous with transmission, and the CLK output is active only while transmitting. To receive a character, then, a character must be transmitted, even if it will be ignored by the slave device(s). The program sends hex FF, which is not in the 25C160's command set, as this throw-away "dummy" character.

Note: Each M16C/62 UART supports Continuous Receive Mode in which the CLK output is triggered by reading the receive buffer as well as by writing the transmit buffer; because Continuous Receive Mode is not required for SPI-compatible operation, it is not used in this example.

3.5 Operation

Figure 3 shows an oscilloscope capture of the M30624 reading the 25C160's status register: The M30624 sent the command 0x05 and the 25C160 returned 0x72. The four traces show, from the top, P73 (CHIP_SELECT), P72/CLK2, P70/TXD2, and P71/RXD2. The relationship of TXD2 and RXD2 to CLK2 verifies SPI Mode 3 compatibility. The rising edges of TXD2 are slightly curved because P70 is an open-drain output with an external pull-up resistor, but the rise time is fast enough to meet the setup time of the 25C160. Although RXD2 shows a small spike when the 25C160 begins driving, near the center of the trace, its value is stable well before the rising edge of CLK2. At the right-hand side of the trace, the result of controlling CLK2 via CKPOL is seen.

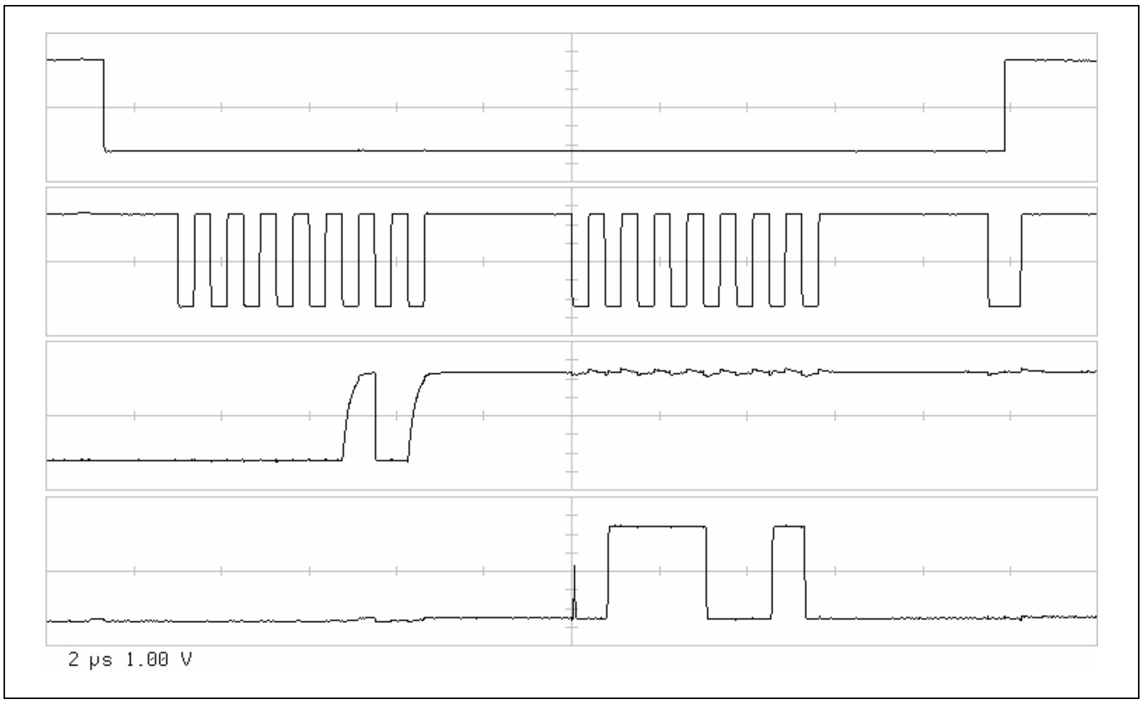

Figure 3 Oscilloscope Capture of an MCU-EEPROM Interaction

4.0 Conclusion

The M16C/62 microcontroller may be used as an SPI-compatible master device for communicating with slave peripheral devices. In synchronous mode, the MCU's UARTs support SPI Modes 1 and 3. As in any multichip application, the designer must satisfy the handshaking and timing requirements of the peripheral devices. The versatility of the M16C/62 assists in these interfacing tasks.

5.0 Reference

Renesas Technology Corporation Semiconductor Home Page

http://www.renesas.com

E-mail Support

support_apl@renesas.com

Data Sheets

- M16C/62 datasheets, 62aeds.pdf

6.0 Software Code

The example program was written to run on the MSV1632 Starter Kit but could be modified to implement in a user application. The program is written in C (the NC30 Compiler), with assembler used for the code executing out of RAM.

```
//****************************************************************************
//
//  Name: spi.c
//
//  Description:  M16C/62 SPI interface example
//
//              This program uses UART2 in clock-synchronous mode to
//              communicate with an SPI peripheral, a 25C160 2Kx8 EEPROM
//              from Xicor or Microchip.  UART2 is polled in this program.
//              The EEPROM is wired as follows:
//                pin 1  CS'    P73
//                    2  SO     P70/TXD2
//                    3  WP'    Vcc
//                    4  Vss    Vss
//                    5  SI     P71/RXD2
//                    6  SCK   P72/CLK2
//                    7  HOLD'  Vcc
//                    8  Vcc    Vcc
//
//              After initialization, this program
//                1) performs several access tests
//                2) reads the entire contents of the EEPROM into RAM
//                3) alters a portion of the RAM buffer's contents
//                4) writes a portion of the altered values to the EEPROM
//                5) re-reads the EEPROM into RAM (buffer should show no
//                   changes)
//
//  Author:  Kevin Clem
//
//  Copyright 2003 Renesas Technology Corporation, Inc.
//
//  All rights reserved
//
//============================================================================
//  $Log:$
//****************************************************************************

#include "sfr62.h"
#include "25C160.h"


#define CHIP_SELECT p7_3

char RAM_array[ EEPROM_SIZE ];
```

```
//------------------------------------------------------------------------------
// UART support routines
//

void uart_tx( register char c )
{
  while( !ti_u2c1 );          // wait for tx buffer to be empty
  u2tbl = c;                  // write the character to tx buffer
}

char uart_rx( void )
{
  uart_tx( 0xFF );            // dummy tx to force a rx
  while( !ri_u2c1 );          // wait for rx buffer to be full
  return( u2rbl );            // return the character from rx buffer
}


void uart_enable_rx( void )
{
  while( !ti_u2c1 );          // wait for tx buffer to empty
  while( !txept_u2c0 );       // wait for tx register to empty
  re_u2c1 = 1;                // enable rx
}


void uart_begin( register char cmd )
{
  CHIP_SELECT = 0;            // enable EEPROM
  te_u2c1 = 1;                // enable tx
  uart_tx( cmd );             // send the command
}
void uart_end( void )
{
  while( !ti_u2c1 );          // wait for tx buffer to empty
  while( !txept_u2c0 );       // wait for tx register to empty
  re_u2c1 = 0;                // disable rx
  te_u2c1 = 0;                // disable tx
  ckpol_u2c0 = 1;             // drive CLK2 low
  CHIP_SELECT = 1;            // bring CS' high
  ckpol_u2c0 = 0;             // let CLK2 idle high
}

//------------------------------------------------------------------------------
// EEPROM support routines
//

char EE_read_status( void )         // read status register
{
  char status;

  uart_begin( RDSR );
  uart_enable_rx();
  status = uart_rx();
  uart_end();
  return( status );
}


void EE_write_status( char new_status )   // write to status reg (clears WEL)
{
  uart_begin( WRSR );
  uart_tx( new_status );
  uart_end();
}


void EE_write_enable( void )              // set WEL
{
  uart_begin( WREN );
  uart_end();
}


void EE_write_disable( void )             // clear WEL
{
  uart_begin( WRDI );
  uart_end();
}

char EE_read( unsigned int addr, unsigned int count )
{
  if( addr >= EEPROM_SIZE || count == 0 || count > EEPROM_SIZE )
    return( 1 );

    uart_begin( READ );
    uart_tx( addr >> 8 );           // send address MSB
    uart_tx( addr & 0xFF );         //   and LSB
    uart_enable_rx();
}
```

```
do {
    if( ( addr & 0x000F ) == 0 ) {
      uart_end();
      while( EE_read_status() & 0x01 );    // check WIP
      EE_write_enable();             // set WEL
      uart_begin( WRITE );
      uart_tx( addr >> 8 ); // send address MSB
      uart_tx( addr & 0xFF );              //   and LSB
    }
    uart_tx( RAM_array[ addr ] );
  } while( --count > 0 && ++addr < EEPROM_SIZE );
  uart_end();
  while( EE_read_status() & 0x01 );        // check WIP
  return( 0 );
}

//-------------------------------------------------------------------------------
// initialization & main routines
//

void initialize( void )
{
  // system clock initialization
  prc0 = 1;                  // unprotect cm1&0
  cm0  = 0x18;                      // div-by-1 mode
  prc0 = 0;                  // protect all

  // GPIOs
  pur2 = pur1 = pur0 = 0xFF;        // enable pull-ups on all ports
  p7   = 0xFF;                      // P73(CHIP_SELECT) & P72/CLK2 are high
  pd7  = 0x0D;                      // P73, P72/CLK2, P70/TXD2 are outputs

  // configure UART2 to be compatible with SPI Mode 3
  u2smr2 = 0x00;             // not I2C mode
  u2smr  = 0x00;             //   ditto
  u2mr   = 0x01;             // internal clock, synchronous mode
  u2c0   = 0x90;             // msb first, CKPOL=0, no CTS/RTS, clock is f1
  u2c1   = 0x00;             // no data reverse, rx & tx disabled
  u2brg  = 0x05;             // 16 MHz / 3 MHz - 1 (round up)
}


void far main( void )
{
  int i;

  initialize();                     // initialize system clock, ports, & UART2

  i = EE_read_status();             // indeterminate, i = 'bx111_xxx0

  EE_write_enable();         // set WEL
  i = EE_read_status();             // indeterminate, i = 'bx111_xx10

  EE_write_status( 0xFF );  // clears WEL
  while( ( i = EE_read_status() ) & 0x01 );      // check WIP; at exit, i = 0xFC
  EE_write_enable();         // set WEL
  i = EE_read_status();             // i = 0xFE

  EE_write_status( 0x00 );   // clears WEL
  while( ( i = EE_read_status() ) & 0x01 );      // check WIP; at exit, i = 0x70

  EE_write_enable();         // set WEL
  i = EE_read_status();             // i = 0x72

  EE_write_disable();               // clears WEL
  i = EE_read_status();             // i = 0x70

  EE_read( 0, EEPROM_SIZE );        // copy EEPROM into RAM_array
  i = EE_read_status();             // i = 0x70

  for( i = 3; i < 39; ++i ) // alter RAM buffer
    RAM_array[ i ] ^= ( 'a' - 'A' );

  EE_write_enable();         // set WEL
  i = EE_read_status();             // i = 0x72

  EE_write( 5, 19 );         // copy portion of RAM_array to EEPROM
  while( ( i = EE_read_status() ) & 0x01 );      // check WIP; at exit, i = 0x70

  EE_read( 0, EEPROM_SIZE );        // copy EEPROM into RAM_array

  while( 1 );                       // all done
}
Required header file “25C160.h”:

// Definitions for 25C160 EEPROM

#define EEPROM_SIZE 2048    // in bytes

// Instruction set
#define WRSR  0x01   // Write the status register
#define WRITE 0x02   // Write data to memory array
#define READ  0x03   // Read data from memory array
#define WRDI  0x04   // Reset the Write Enable Latch (disable writes)
#define RDSR  0x05   // Read the status register
#define WREN  0x06   // Set the Write Enable Latch (enable writes)


// 76543210             Status Register Bit Definitions
// ||||||||
// |||||||+- WIP (RD only): Write In Process   (1=write in progress)
// ||||||+-- WEL (RD only): Write Enable Latch (see below)
// ||||||
// |||||+--- BP0 (R/W): Block         BP1  BP0 | write-protected addresses
// ||||+---- BP1 (R/W):   Protection   0    0  |    none
// ||||                                0    1  | upper 1/4 (0x600 - 0x7FF)
// |+++----- unused                    1    0  | upper 1/2 (0x400 - 0x7FF)
// |                                   1    1  |    all    (0x000 - 0x7FF)
// |
// +-------- WPEN (R/W): Write Protect Enable (see below)
//
//
//          WP'         |  Protected  Unprotected   Status
//   WPEN   (pin)  WEL  |   blocks      blocks     Register
//   ----   -----  ---  +  ---------  -----------  ---------
//    X       X     0   |  Protected   Protected   Protected
//    0       X     1   |  Protected   Writable    Writable
//    1      Low    1   |  Protected   Writable    Protected
//    X      High   1   |  Protected   Writable    Writable
//
// Notes:
// 1) WEL is cleared at power-up and on successful completion of WRDI,
//    WRSR, or WRITE commands.
// 2) CS' must be set high for writes to occur.
// 3) SCK must be low when CS' goes high.
```